Introducción a Rust

Técnicas de Programación Concurrente

Índice

- ¿Por qué Rust?
- Sintaxis
- Ownership
- Threads
- Testing

¿Por qué Rust?

- Objetivos de Rust: velocidad, seguridad y **concurrencia**.
- Se siente como un lenguaje expresivo de alto nivel, mientras que alcanza muy alta performance.
- Adhiere al principio de "zero-cost abstractions".
- Palabras de la comunidad: "empoderar a todos", extremadamente rápido, concurrencia sin miedo, retrocompatibilidad.

¿Por qué Rust?

Seguridad

- Prevenir el acceso a datos/memoria inválida, en tiempo de compilación (memory safe sin costo en tiempo de ejecución) (**buffer overflow**).
- No existen los "punteros sueltos" (**dangling pointers**), las referencias a datos inválidos son descartadas.
- Previene condiciones de carrera sobre los datos al usar concurrencia.

Productividad

- ayudas del compilador, tipos de datos sofisticados, pattern matching, etc.
- Herramienta de construcción incorporada con gran biblioteca de paquetes (cargo)

¿Por qué Rust?

```
error: unknown start of token: \u{2212}
 --> examples/funciones.rs:1:22
  |
1 | fn sumar_uno(a: i32) -> i32 {
  |                      ^
  |
help: Unicode character '-' (Minus Sign) looks like '-' (Minus/Hyphen), but it is not
  |
1 | fn sumar_uno(a: i32) -> i32 {
  |                      ~
```

Hola Mundo!

```
fn main() {
    println!("hola mundo!");
}
```

Tipos de datos

Primitivos

- i8, i16, i32, i64
- u8, u16, u32, u64
- f32, f64
- usize, isize (depende del target)
- bool (1 byte)
- char (Unicode - 4 bytes)

Arrays

- [u8; 3]
- Slices: arr[3..5]

Tuplas

- (char, u8, i32)

Dinámicos std lib

- Vec<T>, Option<T>
- String

Variables

```
// Las variables son inmutables por default
let t = true;

// Para hacerlas mutables usamos mut
let mut punto = (1_u8, 2_u8);
punto = (4, 3);

// Los tipos pueden ser inferidos como en los ejemplos anteriores,
// o explícitos. La asignación se denomina "binding"
let numero: i32 = 42;
```

Funciones

```
fn sumar_uno(a: i32) -> i32 {
    a + 1
}

// Closure
let plus_one = |a| { a + 1 }

// Closure como parámetro
fn map42(mapper:fn(i32) -> i32) -> i32 {
    mapper(42)
}
```

Structs

```
// Campos con nombre
#[derive(Debug)]
struct Persona {
    nombre: String,
    apellido: String
}

// O posicionales
struct NumeroImaginario(f64, f64);
```

Structs - Métodos

```
impl NumeroImaginario {
    // "Método estático"
    fn new(r:f64, i:f64) -> NumeroImaginario {
        NumeroImaginario(r, i)
    }

    fn modulo(&self) -> f64 {
        (self.0*self.0 + self.1*self.1).sqrt()
    }
}
```

Traits

```
trait MagnitudVectorial {
    fn norma(&self) -> f64;
}


impl MagnitudVectorial for NumeroImaginario {
    fn norma(&self) -> f64 {
        self.modulo()
    }
}


fn max(v1:&MagnitudVectorial, v2:&MagnitudVectorial) -> f64 {
    v1.norma().max(v2.norma())
}
```

Enums

```
enum Palo {
    Oro,
    Copa,
    Espada,
    Basto,
}

let palo: Palo = Palo::Oro;
```

Enums

```
enum Message {
    Fire,
    Move { x: i32, y: i32 },
    Say(String),
}
```

Pattern Matching

```
fn print_message(m:Message) {
    match m {
        Message::Fire => println!("Fire"),
        Message::Move{ x:_, y} if y > 10 => println!("Corre hacia arriba"),
        Message::Move{ x, y} => println!("Se mueve hacia {}, {}", x, y),
        Message::Say(msg) => println!("Dice: {}", msg),
    };
}
```

Option y Result

```
enum Option<T> {
    Some(T),
    None
}
fn dividir(num: f64, den: f64) -> Option<f64> {
    if den == 0.0 {
        None
    } else {
        Some(num/den)
    }
}
```

Option y Result

```
enum Result<T, E> {
    Ok(T),
    Err(E)
}
fn contar_lineas(path:String) -> Result<u64, String> {
    let file = File::open(path);
    if !file.is_ok() {
        return Err(String::from("error al abrir archivo"))
    }
    Ok(42)
}
```

Option y Result

```
let d = dividir(3., 0.);
d.unwrap() * 3 // pum!
d.expect("dividió por 0") * 3 //pum!
d.map(|n| n*3)
```

Option y Result

```
fn div_mul(n:f64, d:f64, m:f64) -> Option<f64>
{
    let d = dividir(n, d)?;
    Some(d * m)
}
```

Ownership

```
fn envejecer(mut p:Persona) {
    p.edad += 20;
    println!("{:?}", p)
}
fn main() {
    let mut ariel = Persona { edad: 37 };
    println!("{:?}", ariel);
    envejecer(ariel);
    println!("{:?}", ariel);
}
```

Ownership

```
fn envejecer(p:&mut Persona) {
    p.edad += 20;
    println!("{:?}", p)
}

fn main() {
    let mut ariel = Persona { edad: 37 };
    println!("{:?}", ariel);
    envejecer(&mut ariel);
    println!("{:?}", ariel);
}
```

Ownership

```
fn main() {
    let p: &Persona;
    {
        let ariel = Persona { edad: 37 };
        p = &ariel;
    }
    println!("{:?}", p);
}
```

Ownership

Motivación: eliminar clases enteras de errores (punteros nulos, uso posterior a liberación, doble liberación, saturación de búfer (buffer overflow), invalidación de iterador, carreras de datos) al restringir programas válidos, sin incurrir en gastos generales de ejecución. Inspiración: C ++ RAII (resource acquisition is initialization)

- Un recurso se inicializa cuando se asigna.
- La inicialización la realizan los constructores de clases.
- La limpieza se realiza mediante destructores de clases.
- Los destructores se llaman automáticamente cuando el objeto sale de su alcance.

Ownership

- Cada valor en Rust tiene una variable, que es llamada su dueño (owner).
- Puede haber un único dueño a la vez. Cambiar de dueño es hacer un **move**
- El dueño puede prestar (borrow) múltiples referencias inmutables y **una única referencia mutable a la vez**.
- Cuando el dueño sale fuera de su scope, el valor es eliminado (dropped - trait Drop).

Heap

```
#[derive(Debug)]
enum List<T> {
    Cons(T, Box<List<T>>),
    Nil,
}
fn main() {
    let list: List<i32> =
        List::Cons(1,
            Box::new(List::Cons(2,
                Box::new(List::Nil))));
    println!("{:?}", list);
}
```

Heap

- Box<str> -> String
- Box<[T]> -> Vec (además es iter)
- Box con múltiples referencias inmutables -> Rc
- Box con múltiples referencias inmutables desde múltiples threads -> Arc

Threads

```
println!("[PADRE] spawneo hijo");
let join_handle = thread::spawn(move || {
    println!("[HIJO] spawnie");
    thread::sleep(Duration::from_secs(2));
    println!("[HIJO] me desperté")
});
println!("[PADRE] esperando hijo");
join_handle.join();
println!("[PADRE] terminó");
```

Testing

```rust
fn add(l: i32, r: i32) -> i32 {
    l + r
}
#[cfg(test)]
mod tests {
    use super::*;
    #[test]
    fn it_adds_i32s() {
        assert_eq!(5, add(3, 2));
    }
}
```

Referencias

- The Rust Programming Language, https://doc.rust-lang.org/book/
- Programming Rust: Fast, Safe Systems Development, 1st Edition, Jim Blandy, Jason Orendorff. 2017.
- Rust in Action, Tim McNamara, Manning. 2020.